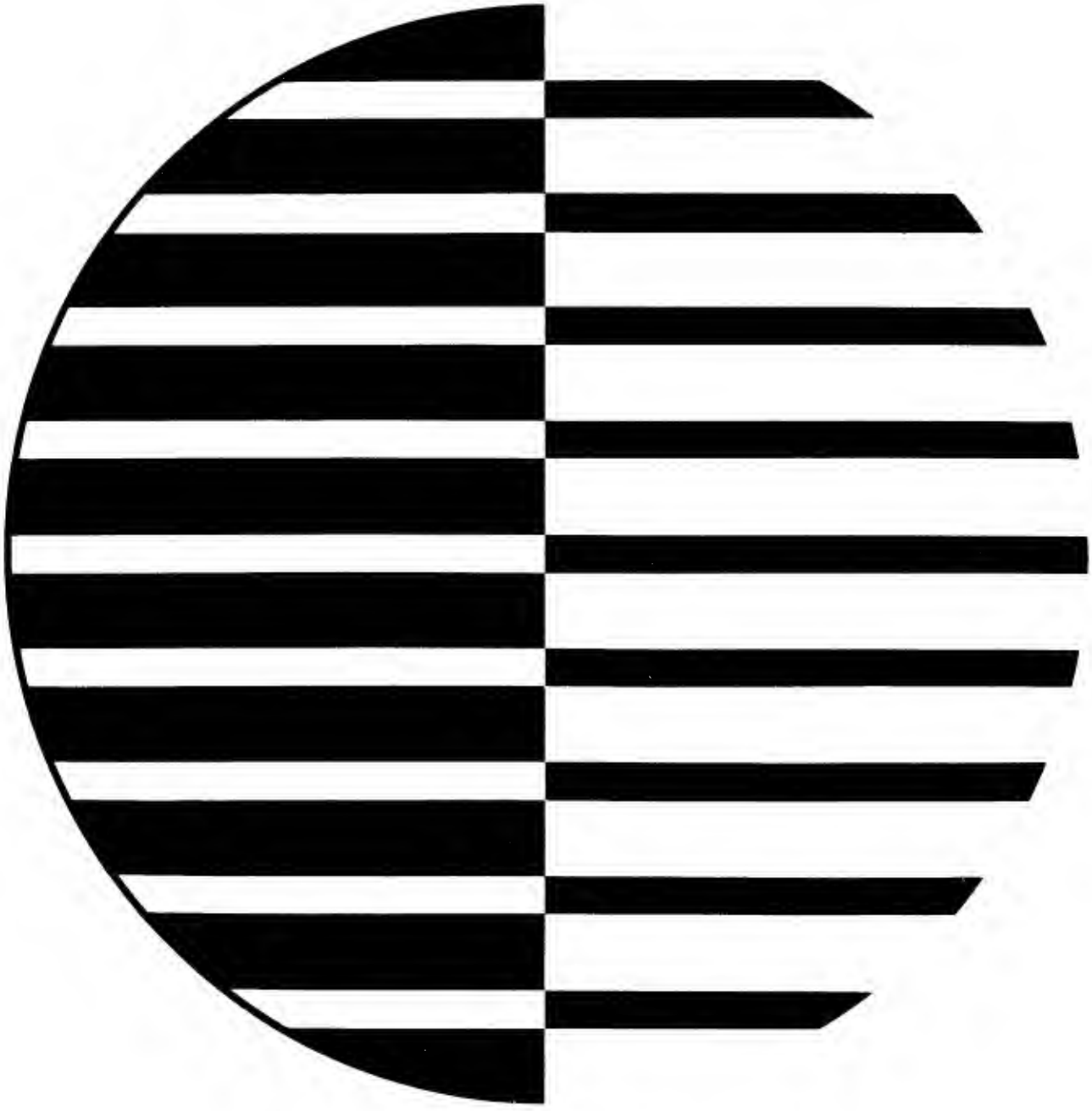
# CONTROL DATA®
# 6400/6500/6600 COMPUTER SYSTEMS
## JOVIAL General Information Manual



## REVISION RECORD

| REVISION | DESCRIPTION |
|---|---|
| A | This is the first edition of this publication. |
| 3-28-69 | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| Publication No. 60252100 | |

Additional copies of this manual may be obtained from the nearest Control Data Corporation sales office.

©1969
Control Data Corporation
Printed in the United States of America

Address comments concerning this manual to:

**CONTROL DATA CORPORATION**
*Documentation Department*
**3145 PORTER DRIVE**
**PALO ALTO, CALIFORNIA 94304**

or use Comment Sheet in the back of this manual

# CONTENTS

# INTRODUCTION 1

JOVIAL (J3) for the CONTROL DATA® 6400/6500/6600 computer systems is a general purpose, procedure oriented programming language. It is particularly appropriate for command and control applications, as well as scientific and business applications. JOVIAL uses conventional English and familiar algebraic and logic notation, has no source code formatting restrictions, and permits comments to be interspersed freely among the language symbols. It provides for the definition and manipulation of a wide range of data values and for control over their structure and storage allocation.

The compiler is designed to produce highly efficient object code and to process source cards rapidly in order to satisfy real time environment requirements and to take advantage of the high speed execution characteristics of the 6000 series computer systems.

## CONFIGURATION

The JOVIAL compiler requires the presence of SCOPE 3, the loader, and system routines. Minimum hardware is that required for the SCOPE 3 operating system, which includes:

1 6400, 6500, or 6600 Computer with 32K Central Memory

1 405 Card Reader with Controller

1 501 Line Printer with Controller

1 415 Card Punch with Controller

2 607 Magnetic Tapes

24 million characters of mass storage on any combination of the following:

854 Disk Pack

865 Drum

6603 Disk File

6638 Disk File

# FEATURES 2

- Independently compilable subprograms
- Subprograms in languages other than JOVIAL
- Compool of data and subprogram declarations
- Library of procedures and functions
- No formatting restrictions on source language
- Capability to intersperse comments with source code
- Capability to embed machine code in JOVIAL program
- Constants and variables of several types:

| | |
|---|---|
| integer | literal |
| floating point | Boolean |
| fixed point | status |
| octal | entry |

- Boolean, arithmetic, and relational operators
- Mixed mode arithmetic
- Partial word (bit and byte) manipulation
- Multi-dimensional arrays
- Variety of file and table structures, including variable size capability
- Automatic or described data packing
- Storage allocation capability
- Formulas and functions as subscripts
- Multiple subroutine exits
- Definition facility
- MONITOR language extension for run-time debugging

# CODE OPTIMIZATION 3

A major objective of the 6000 series JOVIAL compiler is the production of efficient object code. Both machine independent (global) and machine dependent (local) optimization is performed. If desired, global optimization can be suppressed. Some of the techniques used are:

## GLOBAL

- Redundant computation elimination
- Code redistribution
- Value substitution
- Operator strength reductions
- Dead quantity analysis

## LOCAL

- Multiplication and division by shifting
- Arithmetic and type conversions at compile time
- Deletion of extraneous operations
- Special handling of partial word manipulations
- Instruction scheduling
- Expression computation rearrangements
- Register assignment and utilization efficiency operations

# NOTATION 4

In the discussion that follows, a standard notation is used to describe the JOVIAL language forms. This notation is not part of the language; it indicates the order in which the elements appear and the options permitted the user.

| | |
|---|---|
| Uppercase words | JOVIAL words with a predefined meaning for the compiler are written in uppercase letters. These words must be included in the form in which they appear and must be spelled correctly. |
| Lowercase words | Generic terms for classes of language elements are written with lowercase letters. The user supplies the specific elements according to the description of the generic term supplied in the accompanying text. |
| Brackets [ ] | Any word or phrase that can be included in or omitted from a JOVIAL form at the user's option is enclosed in brackets. |
| Braces { } | Optional words or phrases are stacked one above the other and enclosed in braces when only one of the stacked items can be chosen. |
| Punctuation | Any punctuation included in the forms is part of the structure of the form and must be included unless specifically noted otherwise. |
| Space | Wherever one space is shown, the user can supply more than one space. Generally, spaces separate symbols but are not included in symbols; exceptions are noted. |

# LANGUAGE ELEMENTS 5

A JOVIAL program consists of statements and declarations composed from the elements of the language. The elements are symbols and formulas formed from the set of JOVIAL signs.

## SIGNS

The signs, or JOVIAL character set, are:

Letters of the alphabet from A-Z

Digits from 0-9

Marks + - * / . , = ( ) ' $

## BASIC SYMBOLS

Basic symbols have a predefined function and meaning in the language. They are either words formed from two or more letters of the alphabet, or ideograms formed from the set of marks. The words are called primitives and cannot be duplicated.

The basic symbols fall into the following categories:

| | |
|---|---|
| Operators | control the operations to be performed |
| Separators | serve as punctuation |
| Brackets | enclose groups of symbols |
| Functional modifiers | describe particular characteristics of program elements |
| Declarators | introduce declarations |
| Descriptors | single letter codes describe characteristics of declarations |
| Directives | instruct the compiler to perform certain functions |

## USER DEFINED SYMBOLS

Symbols, constructed by the user according to specific rules, fall into three categories: names, constants, and variables.

## NAMES

Names must conform to the following rules:

Contain two or more letters, numerals, or primes

Begin with a letter

Not end with a prime

Not contain two consecutive primes

Not duplicate a primitive

In general, no two names in a program can be identical; however, JOVIAL allows some duplication of names. For instance, names used in status constants may be used elsewhere in the same program; a device name can be duplicated by a statement name, program name, or switch name, and any of these names can duplicate an item name, table name, file name, procedure name or function name.

## CONSTANTS

Constants are specific values that never change during program execution The following kinds of constants are allowed:

| | |
|---|---|
| Integer | Number followed optionally by the letter E followed by a scale |
| Octal | Letter O followed by an octal number in parentheses |
| Floating | Decimal number, including a decimal point, followed optionally by the letter E and a scale |
| Fixed | Floating constant followed by the letter A and a scale |
| Literal | Number followed by the letter H or T followed by a string of signs in parentheses. The number specifies the number of signs in the string; H specifies display code, T specifies transmission code |
| Boolean | Numeral 0 for false or 1 for true |
| Status | Letter V followed by a letter or name in parentheses. Status constants are defined in sets and associated with a status item; their value depends on context. |

## VARIABLES

Variables are named variables, loop variables, or functional modifier variables.

Named Variables These variables are given a simple or subscripted name:

name or ($ name $)

A named variable can be an integer, an octal, floating, or fixed point number; or it can be a Boolean, literal, or status variable.

Loop Variables These are always a single letter used within the context of a FOR loop as an iteration counter. A loop variable can be an integer only.

Functional Modifier Variables Certain variables are introduced by functional modifiers. They are described below according to type:

Integer:

| | |
|---|---|
| BIT ($ index $) (named-variable) | Specifies particular substring of bits in named variable |
| POS (file-name) | Designates position of named file |
| NENT (name) | Number of entries in named variable length table |
| CHAR (floating-variable) | Designates exrad (characteristic) of named floating variable |
| LOC (name) | Starting address in memory of named program, statement, item, or table |

Fixed:

| | |
|---|---|
| MANT (floating-variable) | Designates significand (mantissa) of named floating variable as a signed fixed fractional value |

Literal:

| | |
|---|---|
| BYTE ($ index $) (named-literal-variable) | Designates particular substring of bytes in named variable |

Boolean:

| | |
|---|---|
| ODD { (loop-variable) / (named-numeric-variable) } | Assumes value 1 (true) when variable is odd, the value 0 (false) when variable is even |

Entry:

| | |
|---|---|
| { ENT / ENTRY } (name / ($ index $)) | Specifies contents of entire entry in named table |

## FORMULAS

Formulas express values in the JOVIAL language. Constants and variables are themselves formulas; they can be combined with operators and brackets to form other formulas. A function, which is a call to a procedure returning a single value, is a special class of formula.

### Numeric

Numeric formulas combine numeric constants, functions, and variables with arithmetic operators to produce a single value upon evaluation. Parentheses can be used for grouping symbols and controlling the sequence of evaluation. The brackets (//) and the functional modifier ABS indicate absolute value.

### Literal

A literal formula is a literal or octal constant, a literal variable, or a literal function.

### Boolean

A Boolean formula is either relational or logical. Relational formulas are constants, variables, and formulas, separated by the relational operators:

| | |
|---|---|
| EQ (equal to) | GR (greater than) |
| LQ (less than or equal to) | GQ (greater than or equal to) |
| LS (less than) | NQ (not equal to) |

Logical formulas are Boolean variables, or relational formulas separated by the logical operators: AND, OR, and NOT. Evaluation of a Boolean formula determines whether the formula is true or false.

### Sequential

The value of a sequential formula is a statement name or a close name, and it is designated directly by a sequential operator or indirectly by a switch name. Sequential formulas specify decision points which control the sequence of execution.

## COMMENTS

Comments can be inserted anywhere within a JOVIAL program; a comment is a string of characters enclosed in a pair of double primes. Comments are printed in the listing but are not compiled as code.

# DATA DECLARATIONS 6

Declarations may appear anywhere in the program, but they must precede any reference to the data they define. Names of simple items need not be defined in declarations since they are assumed to be signed whole-word integers.

## DIRECTIVES

Two directives instruct the compiler how to treat data.

### MODE

The default mode for simple items is redefined by mode:

```
MODE item-description [p constant] $
```

Any undefined simple item is defined by the mode directive according to the specified item description; optionally the item is preset to a constant value.

### DEFINE

The user can assign a name to a lengthy expression, make simple additions to the language, or create symbolic parameters:

```
DEFINE name "character-string" $
```

## DECLARATIONS

Data declarations arrange data internally as items, arrays, and tables; externally as files.

### ITEM

Items can be integer, floating, fixed, literal, status, or Boolean.

```
ITEM name item-description $
```

The item description corresponds to the item type. The description can be omitted for integer, floating, fixed, or literal items when a constant of the item type is included to serve as an implicit declaration:

ITEM name constant $

### OVERLAY

The order in which declared data is allocated storage space by the compiler is indeterminate. The user can, however, control allocation of data with the overlay declaration:

$$\text{OVERLAY name}_1 \left[\left\{\begin{matrix}=\\,\end{matrix}\right\} \text{name}_2\right] \ldots \left[\left\{\begin{matrix}=\\,\end{matrix}\right\} \text{name}_n\right] \$$$

When the named items, arrays, or tables are separated by commas they are allocated sequential storage; storage is allocated from the same origin when they are separated by equals signs. Commas and equals signs can be intermixed in the same declaration.

### ARRAY

An array is an arrangement of like items in one or more dimension.

ARRAY name dimension-list item-description [BEGIN constant-list END] $

The item description describes the entire array and each item in the array. The dimension list is a series of one or more integers, each representing a dimension and its value, the number of integers determines the size of the dimension. For instance, the dimension list for a one dimensional array is one integer, for a two dimensional array two integers, and so forth. If a constant list is included, it must correspond to the dimension list.

The entire array is referenced by the array name; individual items are referenced by the array name with a subscript. The subscript is one or more integers corresponding to the number of dimensions; integers are separated by commas and the whole subscript is enclosed in subscript brackets:

array-name ($ $n_1, n_2, \ldots, n_n$ $)

## TABLES

A table is a one dimensional arrangement of one or more entries, each entry having an identical substructure of one or more items. Entries are referenced by subscripting the table name with an entry index; items by subscripting the entry name with an item index.

Tables are declared in two parts: table header and table entry declarations. The table header declaration describes the table as a whole; it specifies the table name, size, structure and packing. The header is followed by a table entry declaration composed of a set of item declarations enclosed in BEGIN END brackets.

Table Size is either variable (V), with the number after V specifying the maximum number of entries, or rigid (R), with the number after R specifying the exact number of entries.

Table Structure is either serial (S) or parallel (P). In a serial table, each item in an entry is stored consecutively. In a parallel table, the first word of each entry is placed in the first block of storage, the second word of each entry in the second block and so forth. A number following P or S indicates the number of computer words needed for each entry.

Table Packing is N for no packing, M for medium packing, or D for dense packing. When N is specified, each item occupies its own word or words in storage and no space is shared. Medium and dense packing allow data to share words.

Table type may be declared as ordinary, defined, or like.

## ORDINARY TABLE

TABLE [name] $\left\{\begin{matrix}V\\R\end{matrix}\right\}$ size [$\left\{\begin{matrix}P\\S\end{matrix}\right\}$] [$\left\{\begin{matrix}N\\M\\D\end{matrix}\right\}$] $

```
BEGIN
   ordinary-entry-declaration
END
```

An ordinary entry declaration contains one or more item declarations of the form:

ITEM name item-description $

optionally followed by:

[one-dimensional-constant-list]

[OVERLAY declarations]

## DEFINED TABLE

The defined table declaration allows the user to allocate the number of words per entry. Packing is assumed to be dense for the table, but it may be specified for individual items or strings in the defined entry declaration. With this declaration, the exact structure of a table can be defined by the user.

TABLE [name] $\left\{\begin{matrix}V\\R\end{matrix}\right\}$ table-size [$\left\{\begin{matrix}P\\S\end{matrix}\right\}$] entry-size $

BEGIN
defined-entry-declaration
END

Defined entry declarations may be item declarations or string declarations.

Item Declaration consists of one or more item declarations:

ITEM name item-description starting-word starting-bit [$\left\{\begin{matrix}N\\M\\D\end{matrix}\right\}$] $

optionally followed by:

[one-dimensional-constant-list]

String Declaration is used to specify more than one occurrence of an item per entry. Each such occurrence is called a bead. The declaration is one or more string declarations of the form:

STRING name item-description starting-word starting-bit
[$\left\{\begin{matrix}N\\M\\D\end{matrix}\right\}$] bead-frequency bead-number $

optionally followed by:

[two-dimensional-constant-list]

### LIKE TABLE

The table named in this declaration has a structure patterned on a previously named and declared table.

TABLE name [$\left\{\begin{matrix}V\\R\end{matrix}\right\}$] [table-size] [$\left\{\begin{matrix}P\\S\end{matrix}\right\}$] [$\left\{\begin{matrix}N\\M\\D\end{matrix}\right\}$] L $

The name of the like table is specified by suffixing a letter or numeral to the name of the pattern table. This suffix is then appended to the names of any items in the pattern table to form item names for the like table. No entry description is included.

## FILES

A file is a sequential string of bits, residing on an external storage device, divided into one or more logical records. A file declaration describes the structure of data in a file.

### FILE DECLARATION

FILE name $\left\{\begin{matrix}B\\H\end{matrix}\right\}$ record-count $\left\{\begin{matrix}V\\R\end{matrix}\right\}$ record-size status-list device-name $

A file is binary (B) or display code (H), and records are variable (V) or rigid (R) in length. The status list is a string of status constants identifying the possible status of the file, such as busy, ready, or error. The device name identifies the particular external device on which the file resides.

# PROCESSING DECLARATIONS 7

Processing declarations describe those parts of a program that are executed only when specifically called: close routines, procedures, and functions. The switch declaration specifies decision points for transfer of control.

## CLOSE

A close declaration describes a closed subroutine without parameters.

```
CLOSE name $
    BEGIN
       declarations and statements
    END
```

This routine is called by the close name, either directly following a GOTO statement or indirectly in a switch call. The compiler provides a transfer around a close routine if it occurs in a block of code. Exit from the routine is normally to the statement that follows the statement calling the close.

## PROCEDURE

A procedure is a closed subroutine that may have input and/or output parameters. A procedure declaration has the form:

```
PROC name
[([formal-input-parameters] [= formal-output-parameters])] $
    BEGIN
       declarations and statements
    END
```

A procedure may not contain procedure declarations or function declarations, but it may contain calls to other procedures or functions.

## FUNCTION

A function is similar to a procedure except that execution of a function must result in a single value. The function name acts as the only output parameter in the function declaration:

```
PROC name [(formal-input-parameters)] $
    ITEM name item-description $
    [declaration-list]
    BEGIN
       declarations and statements
    END
```

The item declaration describes an item with the same name as the function. A function is called by the function name followed by a pair of parentheses enclosing any actual input parameters or enclosing a space if there are no parameters. The function call is effectively the value resulting from execution of the function.

## SWITCH

A switch defines a series of decision points to control the sequence of execution. The switch declaration lists statement, close, or switch names to which control is transferred depending on the value of an item or index. A switch declaration may describe an index switch or an item switch.

Index Switch

```
SWITCH name = (index-switch-list) $
```

The index switch list is a series of statement, close, or switch names separated by commas. The numerical position of each name in the list, starting with zero, provides the index value that determines the name to which control transfers. For instance, a reference to switch-name ($ 1 $) is a reference to the second name in the list.

Item Switch

SWITCH name $\left\{\begin{matrix}\text{item-name}\\\text{file-name}\end{matrix}\right\}$ = (item-switch-list) $

The item switch list consists of constants paired with names of statements, close routines, or switches in the form:

(constant$_1$ = name$_1$ [,constant$_2$ = name$_2$]...[,constant$_n$ = name$_n$])

The current value of item name or file name is compared with constant$_1$, constant$_2$, etc. When a match is found, control passes to the corresponding statement, close, or switch name, otherwise control passes to the statement following the switch call.

# STATEMENTS 8

Statements are the operational units of the JOVIAL language. They describe self-contained rules of computation, specify the manipulation of data, and specify the sequence of program execution conditionally, unconditionally, or both.

## STATEMENT NAMES

A statement can have one or more names, or no name; however, any statement that is referenced in another statement or declaration must have a name. A statement name is followed by a period and a space which is usually followed by the statement but may be followed by another name:

$name_1$. [$name_2$. [...$name_n$.]] statement $

## COMPOUND STATEMENTS

A list of statements enclosed by BEGIN END brackets is a compound statement. Declarations, directives and other compound statements can be included in the statement list. A compound statement is always treated as a single independent statement.

## ASSIGNMENT STATEMENTS

An assignment statement assigns the value of a formula to a variable. An equals sign separates the variable on the left from the formula on the right. The general form is:

[name.] variable = formula $

The variable and formula must agree in type; type may be arithmetic, literal, Boolean, status, or entry. When the statement is executed, the formula is evaluated and its value is assigned to the variable.

## EXCHANGE STATEMENTS

An exchange statement exchanges the values of two variables separated by a double equals sign. No space is permitted between the equals signs. Both variables must be of the same type: arithmetic, literal, Boolean, status, or entry. The general form is:

[name.] variable = = variable $

## CONTROL STATEMENTS

Control statements are used to alter the normal, serial sequence of execution.

### GOTO

The GOTO statement transfers control to a statement designated in a sequential formula. The sequential formula is a statement, close, or switch name.

[name.] GOTO sequential-formula $

### IF

An IF clause is evaluated, and the associated statement is executed only if the Boolean formula in the IF clause is true.

[name.] IF Boolean-formula $ [name.] statement $

### IFEITH ORIF

This variant of the normal IF statement provides a selection of possible statements to be executed depending on the evaluation of Boolean formulas.

[name.] IFEITH Boolean-formula$_1$ $ [name.] statement$_1$ $

[name.] ORIF Boolean-formula$_2$ $ [name.] statement$_2$ $

:

[name.] ORIF Boolean-formula$_n$ $ [name.] statement$_n$ $

END

The Boolean formulas are evaluated in order until one is found to be true, and its associated statement is executed. If the latter statement does not contain a GOTO statement, control then transfers to the statement following END. If no Boolean formula is true, control passes immediately to the statement following END.

### FOR

With the FOR statement, the user can set up an automatic program loop. It sets a counter to an initial value which is automatically incremented or decremented following execution of one or more associated statements until a terminal value is reached. A FOR statement is a complex statement defined as a FOR clause followed by a simple or compound statement:

[name.] FOR letter = initial-value [,increment] [,terminal-value] $

[name.] statement $

The letter is the loop variable or index whose value determines whether the statement following the FOR clause will be executed. Initial value, increment, and terminal value are positive or negative numeric formulas. If only the initial value is specified, the FOR clause is a simple assignment of that value to the loop variable. If the terminal value is omitted, an explicit GOTO must be included in the statement following the FOR clause. The functional modifier NENT (number of entries) can be used as the initial or terminal value when looping through a table.

Several FOR clauses can activate separate loop variables for the same statement. Also FOR clauses can be nested; that is, a compound statement following one FOR clause can include another FOR clause, and so forth.

The FOR ALL clause may be used to loop through a table:

[name.] FOR letter = ALL ( $\left\{\begin{matrix}\text{table-name}\\\text{entry-name}\end{matrix}\right\}$ ) $

This clause assumes the initial value is NENT for the particular table or entry, the increment is -1, and the terminal value is 0.

### TEST

This statement is used to test explicitly a loop variable from within the compound statement following the FOR clause. If the statement contains only one active loop variable. the letter can be omitted.

[name.] TEST [letter] $

## I/O STATEMENTS

Data in the form of logical records can be transmitted between main storage and external storage, and files can be positioned with the input/output statements. One input/output statement is required for the transmission of each logical record.

### INPUT

Three statements govern input:

OPEN INPUT file-name [input-operand] $

OPEN INPUT activates the file named for input, and if there is no operand, it rewinds the file to logical record zero. If an operand is used, the file is opened, a record is read in, and the file is positioned to logical record one (the second record in the file).

INPUT file-name input-operand $

This statement transfers a record of data from an open input file to main storage.

SHUT INPUT file-name [input-operand] $

This statement closes the named file. If an operand is included, data is transferred to storage before the file is closed. An input operand can be a variable, array name, table name, or entry name.

### OUTPUT

Output files are opened and closed, and data is written on the files by commands analagous to the input statements. An output operand can be any of the input operands plus a constant.

OPEN OUTPUT file-name [output-operand] $

OUTPUT file-name output-operand $

SHUT OUTPUT file-name [output-operand] $

### FILE POSITION

The POS statement positions a file to the start of record 0, record 1, up to record n-1, where n is the number of records in the file. The numeric formula is any positive integer which is less than the number of records in the file.

POS (file-name) = numeric-formula $

numeric-formula = POS (file-name) $

## PROGRAM CONTROL

Statements that control the structure of the program include procedure and function calls, RETURN and STOP statements.

### CALL STATEMENTS

A procedure is called by:

name [([actual-input-parameters]) [= actual-output-parameters])] $

A function is called by:

name ([actual-input-parameters]) $

In both cases, the name must be identical to the name specified in the PROC declaration describing the procedure or function, and the actual parameters must agree with the formal parameters in number, type, and order.

A close routine is called simply by its name. When a close name is included in a parameter list, is must be followed by a period.

### RETURN

A close routine, function, or procedure contains an automatic exit to the next statement following the call statement. The RETURN statement allows the user to transfer to that automatic exit from an earlier point in the routine.

```
RETURN $
```

### STOP

The STOP statement halts the sequence of execution; it usually indicates the end of the program. If the program is restarted and statement name is included, control transfers to the named statement.

```
STOP [statement-name] $
```

### DIRECT/JOVIAL

The DIRECT JOVIAL brackets permit the user to include assembly language code in the midst of a JOVIAL program.

```
DIRECT
   assembly code
JOVIAL
```

An ASSIGN statement provides access to variables in the JOVIAL program from within the assembly code. To move the contents of the named variable to the accumulator:

```
ASSIGN A (constant) = named-variable $
```

To move the contents of the accumulator to the named variable:

```
ASSIGN named-variable = A (constant) $
```

The constant is a code defining the particular register and the numeric form of the value in the register.

## MONITOR

This statement traces the flow of execution through designated names and prints the current value of designated variables

[name.] MONITOR [(Boolean-formula)] name-list $

The name list contains item, function, statement, switch, close or procedure names. The value of an item or function is printed whenever it appears in an exchange statement or to the left of an assign statement during execution. A statement, switch, close, or procedure name is printed whenever it is passed or called during execution. If the Boolean formula is present, it is evaluated each time a monitored name is encountered, and only if the formula is true, is the name or value printed.

# PROGRAM STRUCTURE 9

A JOVIAL program is a set of statements and declarations to solve a user's particular processing problem. It may be a main program or a subprogram; each can be compiled separately. A main program is called by the operating system at the user's request. A subprogram can be called by the main program or by another subprogram but it must be described in a compool compiled with the calling program.

## MAIN PROGRAM

A set of declarations and statements enclosed within START TERM brackets is a main program. The form is:

```
START $
    declarations and statements
TERM [statement-name] $
```

If statement name is included, program execution starts with the specified statement.

## SUBPROGRAM

A subprogram is also contained within START TERM brackets, but it includes the subprogram declaration.

```
START subprogram-declaration $
    declarations and statements
TERM $
```

The subprogram declaration is a special case of the PROC declaration:

```
PROC name
    [([formal-input-parameters] [= formal-output-parameters])] $
```

## COMPOOL

A compool is a communications pool which contains data and subprogram definitions. Any valid JOVIAL data declaration can be included in a compool and the data can be preset; processing declarations may not be included in a compool. Independently compiled subprograms must be defined in a

compool before they can be referenced by another subprogram or a main program. Compool defined subprograms may be written in languages other than JOVIAL. If a name is defined in both a compool and a program which references that compool, the program definition takes precedence.

The compool assembler is an optional part of the JOVIAL compiler called by a control card parameter. The standard compool is automatically available; any other compool must be specifically requested on the control card. Assembled compools are available at compile time only. The number of compools is unlimited, but only one may be referenced during compilation. Each compool can contain any number of common data blocks.

A compool specification is similar to a JOVIAL program except that is contains no statements, only declarations:

```
START $ compool-declarations TERM $
```

Compool declarations are either data or subprogram declarations. Compool defined data is arranged in one or more common blocks:

```
COMMON [block-name] $
    BEGIN
        data-declarations
    END
```

The following brief program finds the number of winning and losing rolls out of 5000 rolls of a pair of dice. A table ALL'ROLLS is preset to a maximum of 5000 initial rolls. The program assumes that a roll of 3, 5, 6, or 10 results in an automatic loss; a roll of 7 or 11 results in an automatic win; any other results are not considered.

When processing is complete, the program exits to SYST. The number of wins is contained in the item WINS, the number of losses in the item LOSSES.

```
START $ "DICE GAME PLAYED FEB 14 1969"

    TABLE ALL'ROLLS R 5000 P 1 $

        BEGIN
            ITEM ROLL'ONE I 60 S P 0 $
        END

    ITEM WINS I 60 S P 0 $

    ITEM LOSSES I 60 S P 0 $

    SWITCH ROLL (ROLL'ONE) = (3 = LOSE, 5 = LOSE, 6 = LOSE, 7 = WIN,
                              10 = LOSE, 11 = WIN) $

PLAY.  FOR X = ALL (ALL'ROLLS) $

        BEGIN "X"

            GOTO ROLL($X$) $
            TEST X $

WIN.        WINS = WINS + 1 $
            TEST X $

LOSE.       LOSSES = LOSSES + 1 $

        END "X"

        GOTO SYST $

TERM PLAY $
```

## COMMENT SHEET

CONTROL DATA CORPORATION

TITLE: 6400/6500/6600 JOVIAL General Information Manual

PUBLICATION NO. 60252100 REVISION A

Control Data Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

CUT ON THIS LINE

General comments:

FROM NAME: _____ POSITION: _____

BUSINESS
ADDRESS: _____

_____

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

**BUSINESS REPLY MAIL**
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

**CONTROL DATA CORPORATION**
*Documentation Department*
**3145 PORTER DRIVE**
**PALO ALTO, CALIFORNIA 94304**

CUT ON THIS LINE

FOLD

FOLD

STAPLE

STAPLE


► ►CUT OUT FOR USE AS LOOSE-LEAF BINDER TITLE TAB

JOVIAL GENERAL INFORMATION MANUAL

CONTROL DATA CORPORATION

CORPORATE HEADQUARTERS, 8100 34th AVE. SO., MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

Pub. No. 60252100

Litho in U.S.A.